SICS Research Report
R89004

ISRN : SICS-R—89/04--SE
ISSN : 0283-3638

The Cache Coherence Protocol of the Data Diffusion Machine

by

Erik Hagersten and Seif Haridi

1989

Swedish Institute of Computer Science
Box 1263, S-164 29 KISTA, SWEDEN

SICS/R-89/8904

The Cache Coherence Protocol of the Data Diffusion Machine

by
Erik Hagersten and Seif Haridi

SICS research report
R89004
ISSN 0283-3638

SICS/R-89/8904

THE CACHE COHERENCE PROTOCOL OF THE DATA DIFFUSION MACHINE

Erik Hagersten and Seif Haridi

Swedish Institute of Computer Science
Box 1263, S-164 28 Kista, Sweden

May 1989

Abstract

The Data Diffusion Machine (DDM) is a scalable shared virtual memory multiprocessor in which the location of a datum in the machine is completely decoupled from its virtual address. In particular, there is no distinguished home location where a datum must normally reside. Instead data migrates automatically to where it is needed, reducing access times and traffic.

The hardware organisation consists of a hierarchy of buses and data controllers linking an arbitrary number of processors each having a large set-associative memory. Each data controller has a set-associative directory containing status bits for data under its control. The controller supports remote data access by "snooping" on the buses above it and below it. The data access protocol it uses provides for the automatic migration, duplication and replacement of data while maintaining data coherency.

The machine is scalable in that there may be any number of levels in the hierarchy. Only a few levels are necessary in practice for a very large number of processors. Most memory requests are satisfied locally. Requests requiring remote access generally cause only a limited amount of traffic over a limited part of the machine, and are satisfied within a small time that is logarithmic to the number of processors. Although designed particularly to provide good support for the parallel execution of logic programs, the architecture is very general in that it does not assume any particular processor, language or class of application.

[1] A different version of this paper is to appear in the Proceedings of the Parallel Architectures and Languages Europe Conference, PARLE, 1989.

1 INTRODUCTION

Message-passing machines and shared-memory machines are the two main classes of parallel (MIMD) computer, and are generally considered to be quite distinct. Message-passing machines typically have many processors with large private memories, linked together by a communications network. Shared-memory machines typically have only a limited number of processors with small private memories or caches, connected by a common bus to a large, physically shared, memory. Message passing machines usually require software to view memory access and communication with other processors as quite separate mechanisms. Software often simulates a form of shared virtual memory, by translating references to remote objects into appropriate messages. Shared-memory machines, on the other hand, usually support shared virtual memory directly, thereby allowing software to achieve communication implicitly through memory access, but require some locking mechanisms to support this. Message-passing machines are generally **scalable** to arbitrary numbers of processors, whereas in shared-memory machines the shared bus and memory is a bottle-neck, placing a limit on the number of processors that can be attached. However, message-passing machines place a much heavier burden on software to partition the computation effectively, and so the scalability of the hardware is only useful insofar as the software can keep communication to a minimum.

The DDM is like a message-passing machine in that memory is distributed and the machine is scalable to an arbitrary number of processors. The DDM is like a shared-physical-memory machine in that it supports a shared address space and processors are connected via buses. The key idea behind the DDM, which distinguishes it from both message-passing machines and shared memory machines, is that the location of a data item in the machine is completely decoupled from its virtual address.

The design of the DDM is based on the following considerations. Where a piece of data resides is not really relevant to the software. Ideally, the physical location of data should be transparent to the software. All the software needs is some means of identifying each data item, which is just the virtual address. Rather than have software control the physical placement of data, this should be taken care of automatically by hardware. Thus virtual addresses should be mapped into physical addresses in a totally flexible manner. The mapping should be dynamic, allowing data to migrate to where it is most needed. It may be desirable to have multiple copies of a particular data item, but they will all share the same virtual address. To summarize, from a software point of view there will be of a number of processes sharing data that is arranged logically in a single virtual address space; from a hardware point of view, processes will be mapped into processors and virtual addresses into physical addresses in such a way that most of a processor's memory accesses can be satisfied by its local memory. In other words, the data structure that the software sees will distribute itself automatically over the machine in such a way as to reduce data access times and minimize data traffic.

The DDM was motivated by our work on logic programming execution models and represents our ideas on how these models can best be supported by hardware. The design, however, is very general in that it does not assume any particular kind of processor, language or application. We feel this is very important if the machine is to gain practical acceptance, and is an important factor in the commercial success of machines such as the Sequent. It should be noted that software designed for conventional shared-memory machines can run without change on a DDM.

The remainder of the paper is organized as follows. In the first section we describe the main feature of the hardware organization. The next section is an introduction to the protocol. Next is a discussion over the need for and implementation of replacement, followed by some remarks of the hardware requirements of the machine. Next we analyze performance characteristics of the machine, and compares it with other architectures. We conclude the paper by bringing up various other issues and a summary of the main novel features of the design. At the end of the paper

Figure 1: The Data Diffusion Machine

protocol tables defining the protocols used is to be found.

2 OVERVIEW OF THE ARCHITECTURE

The machine is hierarchical (see fig. 1). At the tips of the hierarchy are processors each with a large local memory (possibly accessed via a conventional cache). The memory contains an image of some part of the global virtual address space. The memory is set-associative, and is organized like a (very large) cache, but it should be emphasized that this is the sole form of main memory in the machine. The memory is connected via a memory controller to a local bus. The local bus connects a cluster of similar configurations of processor, cache, memory and controller. The local bus may itself be connected via a controller to a higher bus, and so on up the hierarchy. The higher level controllers each have access to a directory of status information, and are termed directory controllers. The directory is set-associative, and has space for status bits for all the data items in the memories below.

Data are stored in blocks, items. The item size is fairly small, possibly a couple of words, and is to be decided after extensive simulation. Memory and directories views a block as being one unit.

The function of a controller is to mediate between the bus above it and the subsystem below it. Its behavior is a generalization of the "snooping" caches in single-bus shared memory processors. It allows memory requests to be handled as locally as possible, but where a request cannot be handled locally, it is responsible for transmitting that request upward or downward to enable it to be satisfied. The controller has access to a directory which tells it which part of the virtual address space is mapped into the memory of the subsystem below it, and whether any of those virtual addresses are also mapped into memory outside the subsystem. Thus for any virtual address, the controller can answer the questions "Is this item below me?" and "Does this item occur elsewhere (not below me)?".

The controller prevents unnecessary communication from entering and leaving its subsystem. A memory request will not be transmitted outside a subsystem if (1) it is a read of a local item or (2) it is a write to an unshared local item. In particular, this means that if a processor tries to read an

item in its local memory or write an unshared item in its local memory, no external communication is required. Normally, this will cover the vast majority of memory references. A memory request will not be transmitted into a subsystem unless (1) the item resides in the subsystem and (2) the subsystem is selected (only if the item resides in more than one subsystem).

If a subsystem tries to read a nonlocal item, a read request will be propagated as far as is necessary to retrieve a copy of the item, and the item will be marked as shared where necessary. If a subsystem tries to write a shared item, a request will be propagated to erase all other copies of the item, and the item will then be marked as unshared.

If a memory becomes full, data items that are shared elsewhere can be discarded; the machine will select items which are least recently used; if there is no such item, an exclusive item that is least recently used will be moved elsewhere. This is another means by which data tend to reside only where it is being actively used.

The following points should be noted. The data that a processor creates itself will automatically reside in its own memory and will not be copied anywhere else unless another processor requires it. A processor is likely to spend most of its time accessing such data. A processor is not obliged to repeatedly access an item from a remote memory, if the data is initially remote. Instead, remote data tends to migrate to where it is being actively used, and is not tied to some fixed "home" location.

3 INTRODUCTION TO THE DDM PROTOCOLS

A multicache system introduces the cache coherence problem. A item can reside in many caches. On writes, the consistency of the system has to be kept, i.e., at any time a item can only have one value. Single-bus systems use snooping protocols where all caches snoop all transactions on the common bus to maintain consistency. We have developed a hierarchical snooping protocol, where caches (our memories) and directories only snoop the bus above them, i.e., only a small portion of the transactions in the system. Transactions are received from below in parallel with the snooping.

Each item has a state associated with it. The protocol specifies a new state and a transaction to perform based on the current state and the transaction received. The protocol is specified by four state-transition tables at the end of the paper: memory below, memory above, directory below and directory above.

3.1 The Ideal Model

The examples below show the state changes and transitions for **one** virtual address. We will walk through a couple of examples.

There are three stable states in the DDM:

I Invalid. The subsystem does not contain the item.

E Exclusive. This subsystem and no other contains the item.

S Shared. This subsystem and possibly other subsystems contain the item.

Transient states will be introduced when needed.

Figure 2 shows a picture of an initial, two-level system with the item residing in the second of the ten processors, P2. Thus, P2 and all directories directly above it each have the item in state E. Everywhere else the item is in state I (nonexistent).

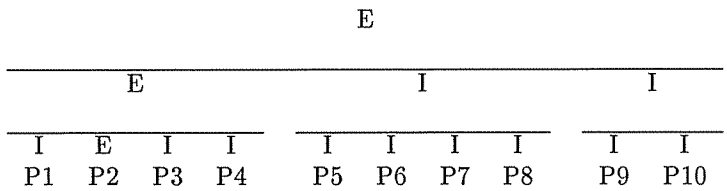
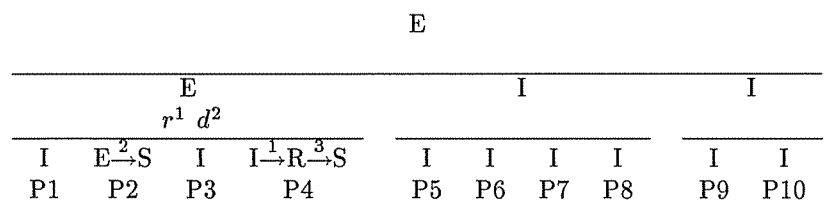


Figure 2: The initial system

The examples assume an ideal system; i.e., memories never get full, buses are never busy, all transactions take the same time, all buses work synchronously and independently of each other, and no buffers are used. State transitions and bus transactions are indexed to indicate in which order they take place.

3.2 Local Read on One Bus

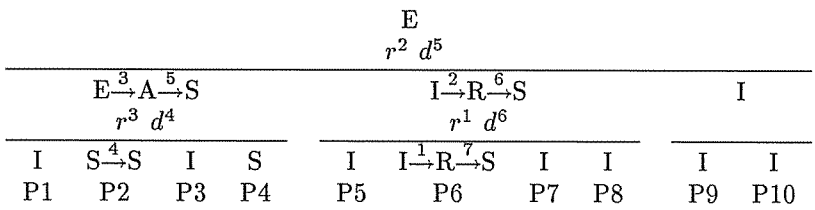


[1] A read by processor 4 to the I item generates a *read* (r). P4 changes its state to R.
[2] The memory with a copy of the item responds with a *data* (d) and changes its state to S.
[3] The requesting memory receives the *data* and puts it in state S.
NOTE: The directory above the bus in the example did not interact in the action described, since the action was local to its subsystem. The directory had the item in state E and anticipated a response from somewhere in its subsystem. No unnecessary bus traffic was generated outside the subsystem.

Figure 3: Local read on one bus

Processors read item in state S or E locally without involving the protocol. In the figure 3, however, P4 tries to read an item in state I. The memory controller generates a *read* transaction on the bus above and temporarily changes the state to R. The need for this state is explained later.

3.3 Nonlocal Read



[1] A read by P6 to an I item generates a *read* and changes the state to R.
[2] The directory detects a nonlocal action and repeats the *read* upwards, changing its state to R.
[3] A directory with state E answers the request by changing its state to A, sending *read* below.
[4] One of the memories, P2, is selected to service the *read*. It stays in S and sends *data* .
[5] The directory in state A has promised to answer. It send *data* above and changes its state to S.
[6] The directory in state R is waiting for the *data*. It changes state to S and sends the *data* below.
[7] The memory in state R is waiting for the *data*. It receives the *data* and changes state to S .
NOTE 1: Many subsystems on a bus may have an item in state S. Letting all of them reply with the *data* would produce unnecessary bus transactions; instead, one is selected in phase 4.
NOTE 2: After phase 3, the return path for data is marked with As and Rs.

Figure 4: Nonlocal read

The *read* and *data* transactions in figure 3 are two separate transactions. The bus is released between the two transactions, while *data* is being prepared. The R state is used to remember which memory asked for the item. Nonlocal reads involve several buses. Here it is even more important not to lock all those buses for the whole read operation. Let's introduce two transient states.

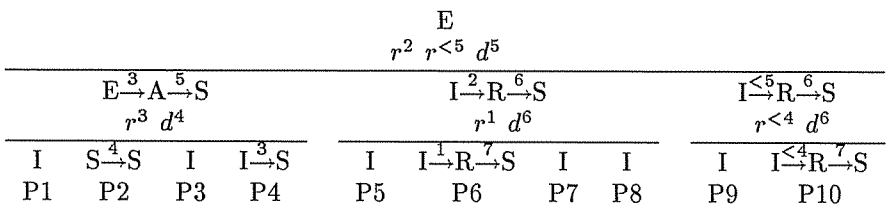
R Reading. This subsystem has sent a *read* request and is waiting for *data* to arrive.

A Answering. This subsystem has promised to answer a *read* request.

State R marks the path of the *read* request on its way up, and state A marks the path on its way down. The *data* uses these states to find its way back to the requesting memory. In figure 4 we start with the final state of figure 3. Now P6 tries to read the item.

3.4 Combining Reads

It is common that many processors try to read the same item; i.e., "the hot spot phenomenon" [8]. The DDM combines read requests to the same item on their way up and on read responses on their way down. Figure 5 shows combining on the way down, or "broadcasting." It differs from figure 4 only in that P10 also reads the item between phase 1 and 4.



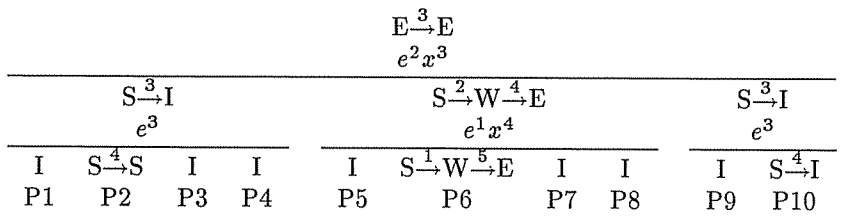
[<4] P10 also reads before phase 4.

[<5] A second *read request* will appear on the top bus generating no extra action.

[6,7] The *data* originally intended for P6 will also be received by P10.

Figure 5: Combining read, broadcasting

3.5 Writing



[1] P6 tries to write to the shared item generates an *erase* (e) and changes state to W.

[2] The directory detects a non local *erase*, changes its state to W and retransmits *erase* above.

[3] Directories in state S detecting receiving an erase from above, change state to I and repeat the *erase* below. The top directory detects a local erase in its subsystem and replies with an *exclusive* (x) below.

[4] The directory in state W receiving a *exclusive* from above knows that it has the only valid copy. It changes to state E and repeats the *exclusive* below. P2 and P10 also gets their copies erased .

[5] P6 receives the good news (*exclusive*), and changes its state to E, carries out the write, and continues with the next instruction.

NOTE: The acknowledge of the *erase* (*exclusive*) is sent when the *erase* reaches the top directory, not when it reaches the memories.

Figure 6: Nonlocal write

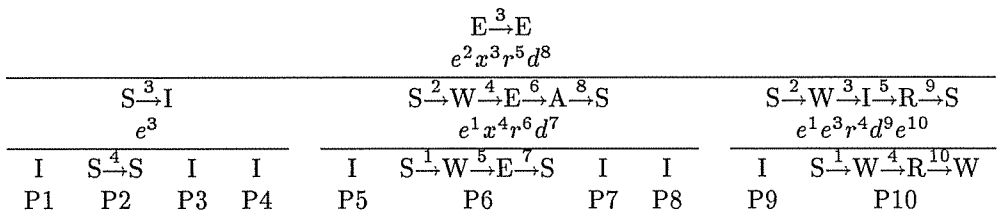
While the main goal of reading is finding and delivering an item, writing involves worrying about the consistency. Processors are only allowed to write to an item in state E. If the item is in state S, all other copies are erased before writing is allowed. A subsystems waiting for all other copies to be erased use a new transient state.

W Waiting. This subsystem is waiting to become exclusive.

Trying to write to an item in state I (write miss) results in a read followed by a write.

Figure 6 starts with the final state of figure 5, P6 writes to a shared item.

3.6 Write race



[1−2] Similar to figure6, both *erase*s work their way up towards the top bus.
[3] The *erase* originating in P6 is the winner and is carried on the top bus. All other directories change their states to I and retransmit the *erase* below.
[4] P10 receives the bad news (*erase*). Instead of just invalidating it starts a read transaction.
[5] P6 becomes the exclusive owner of the item and carries out the write.
[7] The *read* from P10 reaches P6, which changes state to S and sends *data* containing the new value.
[10] The *data* reaches P10 which changes state to W and once more sends an erase. We wish it better luck this time.

Figure 7: Write Race

Race conditions, like two memories trying to write the same item, are solved by the bus arbitration of a real system. Unlike the ideal system in our examples, a real bus arbitrates between the subsystems connected to it. One is selected to carry out the next transaction. The first *erase* to be selected by a bus will cancel any other erases to the same item and wins the write race. The losing memory will generate a new write automatically.

Figure 7 is identical with figure 6, except that both P6 and P10 try to write at the same time.

4 REPLACEMENT IN MEMORY

When a write-miss or a read-miss occurs, a new item will eventually be read into the memory. If the set where the new item is to be stored is full, an old item is chosen to leave its space for the new item; this is called replacement. There are two types of replacement transactions, depending on the state of the item being replaced: moving out and inject.

4.1 Moving Out

If the item selected for replacement is in state S, the replacement is made with a moving-out operation. Moving an item out is more complicated than just throwing away the shared copy. One has to make sure that all copies in the system are not thrown away at the same time. The job is to find another copy of the item.

The replacing memory initiates the *out* by sending an *out* transaction on the bus. The space of the item can now be reclaimed. If the *out* "sees" a subsystem underneath it in either of states S,R,W, or A, it terminates. If not, the directory above will transfer the *out* transaction to the next higher bus and change its state to I, since its subsystem no longer contains the item. The *out* will propagate all the way to the top bus if needed.

4.2 Inject - a Refugee Looking for a New Home

Replacing an item in state E results in an *inject*, or "looking for a new home". A memory with empty space in its set "expresses interest" and, if selected, gives the item a new home. If no memory has space for the *inject*, the transaction is repeated on the next higher bus, where a directory can "express interest". The directory can't "promise" anything, however, and the *inject* might be rejected lower in the system. A counter that follows the *inject* tells how many times an *inject* has been rejected. After the counter has passed a limit, the *inject* will not be given any more tries, and will be sent to the backup storage. A memory that would "like" to send an item directly to the backup storage can do so with an *inject* counter initiated above the limit. The limit of the system might be changed at runtime, according to the system load. The behavior of the *inject* results in a scalable memory, i.e., a single processor running sequential code can use the all the memory in the system.

4.3 Is a Shared Item Always Shared?

An item in state S might actually be either shared or exclusive. The reason for this might simply be that there used to be two copies of the item, but the other one was moved out of its subsystem. This situation is perfectly safe, even if it involves some unnecessary work on the next write. However, it is possible for a directory to detect when one of its clients should be changed from S to E and to change its state. If the item is in state E in the directory and only one subsystem says "got it" to an *out* transaction, the state of the subsystem should be changed to E. It is however unclear whether this will save any work or not.

If the last copy of an item marked with state S is replaced, this will cause an *out* that fails to find another copy in the system. However, when the *out* eventually reaches a directory in state E, it is converted into an *inject*. The *out* carries the data value, which will rarely be used, just to make this conversion possible.

The last two items moving out at the same time is just a special case of the above.

4.4 Having Promised to Answer When the Data Is on its Way Out

A subsystem in state A has "promised" to answer a *read*. Before the *read* reaches the item in the memory, however, the memory has started a replacement, and an *inject* or *out* message now appears on the bus below. This is perfectly fine, since these transactions carry the data value of the item, and the directory can send *data* on the bus above to keep its promise. One has to be careful if it is an *inject* transaction that was accepted by a subsystem on the bus below. The subsystem "believes" that it has the only copy of the item, so the directory can't make use of the data value for answering a read request. Instead, a new read request is sent below.

5 REPLACEMENT IN DIRECTORY

The memory size (size here meaning number of entries) of the directories increases higher up in the hierarchy. However, in order to guarantee space in a directory for all item in its subsystem, it is not enough to just increase the size of its set-associative memory. The memory should also be N ways, where N is the product of the number of memories in its subsystem and their number of ways. In big systems N would be in the range of hundreds. Even if implementable, such memories would be expensive and slow. We have chosen to use directories with smaller sets, called imperfect directories, and to give them the ability to perform replacement.

5.1 The Need for Replacement

The reason for a replacement is a *read* from below to an item in state I, where I is an interpretation of "doesn't exist." A directory with an item in state I receiving a *read* from below is supposed to change its state to R and repeat the *read* on the bus above. However, if the set where the new item is supposed to be is full, an old item has to be chosen and thrown out of the subsystem before the *read* can be carried out.

5.2 Replacement Algorithm

During replacement, the directory has to deal with two different items. The *read* item is called the new item, and the item being replaced is called the old item. The directory starts the replacement with a *leave* transaction of the old item below. One of its subsystems is selected to carry out the *leave*. It sends a *leave* below, and it changes state to the transient state:

L Leaving. This item is about to leave the subsystem.

All the other subsystems erase their copies of the item. When the *leave* finally reaches one memory, the memory changes its state to I and replies with an *up*. The *up* that contains the data value of the item is transferred up through all directories in state L, changing their states to I. When the *up* finally reaches a directory not in state L, it either generates an *out* or an *inject* transaction, depending on the state of the directory. The *read* of the new item has to be repeated while there is space in the directory. The naive way of achieving this is to send an *erase* of the new item immediately after the *leave* of the old item on the bus below the replacing directory. The *erase* will eventually reach the memory initiating the *read*, which will repeat the *read*. Hopefully, there will be enough space this time.

A HW optimization of the above is a small fully associative memory in the directory that stores the item about to leave. This immediately makes space for the new item that carries on with its *read*. The naive *erase* of the new item is needed only if this memory is full. Such memory can be small and still have an impact on performance.

				o^5/i^5						
				$X \xrightarrow{1} X \xrightarrow{5} I$ $l^1 u^4$						
		$S \xrightarrow{2} I$ e^2				$S \xrightarrow{2} L \xrightarrow{4} I$ $l^2 u^3$			$S \xrightarrow{2} I$ e^2	
I	$S \xrightarrow{3} I$	I	I	I	$S \xrightarrow{3} I$	I	$S \xrightarrow{3} I$		$S \xrightarrow{3} I$	I
P1	P2	P3	P4	P5	P6	P7	P8		P9	P10

[1] The directory starts the replacement by sending *leave*. It keeps its state.

[2] The subsystem in the middle is selected to carry on the *leave*. The others merely *erase* their copies.

[3] P6 is selected to carry on the *leave*. It responde by sending *up*. All processors change their states to I.

[4] The *up* reaches the directory in state L, which changes to I and repeats the *up* above.

[5] The replacement is completed. The top directory changes state to I and sends an *out* or *inject* above depending on its state.

Figure 8: Replacing initiated of the top directory

Figure 8 illustrates the above, let's assume our picture of the DDM is a subsystem in a bigger machine. A *read* of another item has forced our top directory to replace the item we are looking

at. The state of our item in the top directory does not matter and is marked with X.

6 HARDWARE REQUIREMENTS

So far we have presented an ideal picture of the machine. This chapter will put the DDM into the scope of the real world.

6.1 DDM Bus

The transactions thus far presented have assumed a bus functionality that cannot be found in existing buses.

- The DDM bus carries split transactions, i.e., a request and its response are two separate transactions.
- Each transaction is tagged with a transaction identifier.
- Each subsystem can say "got it" (a := yes). The answers are ORed together to one signal saying "at least somebody got it" ($a \geq 1$), or "nobody got it" (a=0).
- Some transactions, like *read* require the bus to select one of the subsystems servicing the requests.
- Each bus also has an "emergency brake," used to halt the bus in some cases of full buffers.
- Bus arbitration also differs from normal requirements: it should keep the buffers limited. The next bus master is chosen according to the following priorities:
  1. The subsystem selected in the last transaction.
  2. Directory above.
  3. Round robin between the rest of the subsystems.

The DDM bus can be implemented as a superset of an existing bus, which would save us the work of electrical definitions and give support with tailor-made components targeted for that bus.

6.2 Buffers

Since only one subsystem can send at a time, buffers are needed to avoid deadlocks. Buffers also supply a "rubber band" effect, allowing the subsystems a more even execution. Each subsystem has an output buffer above (OA). Directories also have an input buffer below (IB) and an output buffer below (OB) as shown in figure 9. In some transactions on the bus, killing operations should be performed by the OA buffers below, erasing transactions with a lower priority targeted for the same item. Killing transactions are in priority order: *erase, out, leave.* Bus arbitration limits the OA buffers to a depth of three.

6.3 Flow Control

Since buffer sizes are limited, part of the machine must eventually be halted, using the bus emergency brake, while full buffers are being emptied.

- IB is full either because OA is full or because the directory controller is too busy to empty IB. The directory pulls the emergency brake below. Note that a subsystem being frequently selected to service transactions is more likely to be given a higher priority in bus arbitration.

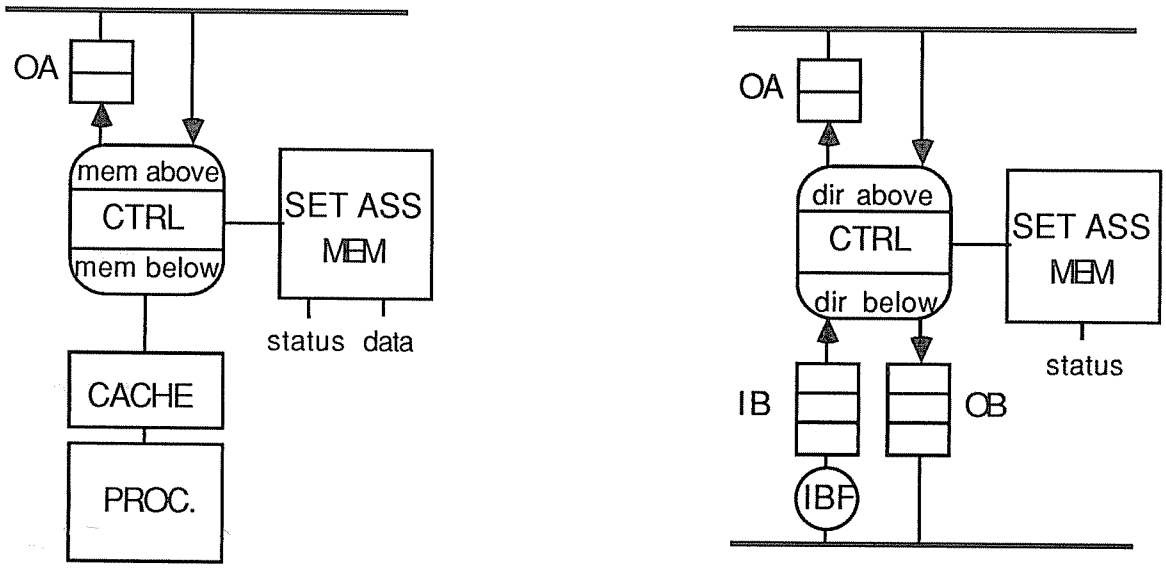


Figure 9: The architecture of the memory and directory

- OB is full. Directory pulls emergency brake above. Note that this is not very likely since the directory above become bus master every second time in the worst case.
- OA never needs to halt the system because of bus arbitration.

In case of IB and OB buffers both being full, both the buses above and below are halted. This is a deadlock situation. A response transaction (erase, data, or exclusive) is chosen from the OB buffer to be carried out on the bus below freeing up space in the OB and releasing the bus above. These transactions are safe to carry out, since they will not generate any additional traffic,

6.4 Top Directory

The number of entries in a directory is greater higher up in the hierarchy. The directory at the very top needs to have one entry for each item in the whole system. Since all item in a top directory are in state E anyhow, the directory and its memory can be replaced by a simple state machine outlined at the end of the paper.

6.5 Broadening Higher Buses

The system described so far has two obvious bottlenecks:

- The size of the directories grows the higher up one gets in the hierarchy. The biggest directories will be found right underneath the top bus. A practical limit to how big these directories can be made limits the size of the system.
- Although most memory accesses will tend to be localized within the machine, the higher level buses may nevertheless demand a higher bandwidth not to become a bottleneck. Snooping in the big directories will make the top bus slower rather than faster.

These two problems both have the same solution: broadening of higher buses. It is possible to split a big directory into two directories of half the size. The directories deal with different halves of the address space (even and odd). The number of buses above is also doubled, each bus dealing with its own address space. This more than doubles the bus bandwidth, since it also brings down the size of the snooping directories. Repeated splits will effectively make a bus as wide as possible, and directories as small as needed. The splitting can be used at any level.

Another way of taking the load off the higher buses is to have a smaller branch factor at the top of the hierarchy than lower down [7]. This solution, however, makes the big directories even bigger.

6.6 Input Below Filter

Not all transactions on a bus need to reach the directory state machine above. A simple filter can eliminate some transactions without using the state in the directory. The input below filter is outline at the end of the paper.

6.7 Size and Overhead of Memories and Directories

An important question is whether it is feasible to store in the higher directories the exact status of all the words below, or whether the higher directories should maintain only lower resolution information based on blocks of words. It appears to be feasible to store exact information. Assuming that memories and directories are 4-way set associative, address space is 1 Gitems, and that each level-1 memory contains 1 M words, each memory or directory at a given level contains the following:

Level 1: 1 M items with 3-bit status, 12-bit key, data value
Level 2: 16 M items with 3-bit status, 8-bit key,
Level 3: 256 M items with 3-bit status, 4-bit key.
Top : No directory needed.

where the keys are the high-order address bits that must be stored to support set-associativity. Thus in a 3-level machine, the overhead per item of storing the extra status information and address keys in both memory and the higher directories is 3+12+3+8+3+4 = 33 bits. With items as small as one word, we are still only doubling the memory requirement in order to provide the DDM's complete flexibility of virtual to physical address mapping. This seems a tolerable price to pay.

6.8 Disks

A data diffusion machine may have one or more disks attached to it. Disks behave as secondary memory subsystems which can hold overflow data. The disks may be distributed all over the system. Each disk will be dedicated to hold some portion of the virtual address space. A *read* reaching the top directory without being serviced will be converted into a *physical-read* finding its way to the disk responsible for that portion of the address space. In a similar way, an *inject* reaching the top directory will be converted to a *physical-write* updating the responsible disk. These conversions are to be found in the top directory state machine, but the *physical-read* and *physical-write* are not included in the protocols.

6.9 Reliability

The probability of faulty hardware increases with the size of a system, and reliability is important for systems of this size. Our plan is to focus on error detection rather than error recovery. Since data do not have home locations and can reside in any part of the system, a detected faulty subsystem can simply be disabled, and the execution rerun. Only a simple error detection and correction code should be used to catch glitch errors.

7 PERFORMANCE CHARACTERISTICS

The data that a processor creates itself will automatically reside in its own memory. As long as no other processor requests the data, the processor that created it can access it without causing any bus traffic. This is likely to cover the vast majority of data accesses. When a item is created by one processor and subsequently accessed by another, the item only needs to be copied once. There is no need to repeatedly access a remote item from its home location, as in most machines. If two nearby processors request the same remote item, one of the processors can obtain it from its neighbour without needing to fetch it twice from the remote location.

A remote read takes at most 4N-2 bus transactions on an N-level machine (2N-1 read requests to pass it up to the topmost bus and down to the data, and the same number of read responses to pass it in the opposite direction). For example, there would be at most 10 transactions on a 3-level machine. To make a item exclusive (in order to perform a write), an erase request goes up to the directory controller level where the item is exclusive; the directory controller acts as a lock and sends erased requests downward. Thus the item becomes exclusive after at most 2N transactions on an N-level machine (N erase requests up and N erased responses down). For example, there would be at most 6 transactions on a 3-level machine.

In general, the protocols have a combining effect on read requests going up, similar to that provided by the IBM RP3 multiprocessor [8], and a broadcast-data effect when read responses are going down, thus eliminating the "hot spot" phenomenon of the RP3. Thus, in a 3-level machine, if one processor has a item and the remaining processors request the same item, more or less simultaneously, all processors will get the item in no more than 10 bus transactions.

In general, remote data accesses only cause traffic within the subsystem concerned. For a read, only buses on the path between the source of the request and the source of the data are involved. For an erase, only buses on paths from the source of the write to copies of the data are involved.

The machine is scalable because theoretically there can be any number of levels in the hierarchy. Note that the data-access protocols are completely independent of the number of levels in the machine or the number of subsystems per bus. In practice there can be quite a few subsystems per bus (e.g., 16), so only a few levels are necessary to support a very large number of processors.

8 BEHAVIOUR IN LOGIC PROGRAMS

An important characteristic of logic programs is that most data items are only written once and thereafter are read-only. Moreover, there is much data that is entirely read-only, including the usually very large volume of program code and the smaller but critically important emulator code (which is in many ways equivalent to microcode).

Shared memory machines such as the Sequent, with no significant local memory (apart from the relatively small caches), will, when executing Prolog, typically waste a large part of the shared bus bandwidth in repeatedly fetching program code and (worse still) emulator code. Ideally a processor should retain a local copy of the emulator and probably a significant part of the program code. Also it should be able to retain within local memory data that has become read-only and

data that is not currently being shared with other processors. The data diffusion machine has these characteristics.

9 LOCKING

Some operations, depending on the nature of the processor, need to be performed atomically. For example if the processor provides a test-and-set instruction this will lead to a read-modify transaction being performed by the memory controller. A read-modify behaves like a write except that before the data item is overwritten the original value of the item is fed back to the processor. With such an implementation of read-modify, together with the general behavior of the machine, it is possible to perform spin-locking locally without generating any traffic, as shown below:

```
Lock(X):
  Start: Flag := Test&Set X;
         if Flag = 0 then Exit;
  Loop:  if X=1 then goto Loop
                else goto Start;
  Exit:

Unlock(X):
           X := 0;
```

where `Flag` is a machine register and `Loop` causes local spinning until `X` is modified. Locking can also very well be built into the DDM protocol with the introduction of a couple of extra states for the memories. A similar locking scheme is presented in [1], where a separate lock directory is introduced in parallel with the cache directory. The cache is intended for a KL1 machine where lock conflicts are rare. Busy wait is used by the waiting processor.

A multiprocessor with thousands of processors, like the DDM, has to explore more fine grained parallelism. In such the support of synchronization between processes will be of higher importance. The behavior of I-structure memory [2], used for synchronization in dataflow, can be achieved with a slight change of our protocol. A locked item can be marked "waited on" in the producer's memory, and marked R in the consumer's (consumers') memory.

10 CONCLUSION

The data diffusion machine is a scalable, shared-virtual-memory multiprocessor where the location of a item in the machine is completely decoupled from its virtual address. In particular, there is no distinguished home location where a item must normally reside. Instead, data migrates automatically to where it is needed, reducing access times and traffic.

The machine is scalable in that there may be any number of levels in the hierarchy. Only a few levels are necessary in practice for a very large number of processors. Most memory requests are satisfied locally. Requests requiring remote access generally cause only a limited amount of traffic over a limited part of the machine and are satisfied within a small time that is logarithmic to the number of processors. Although designed particularly to provide good support for the parallel execution of logic programs, the architecture is very general in that it does not assume any particular processor, language, or class of application.

In future work, we plan to refine the design of the machine to a lower level and to carry out detailed simulations to verify its behavior both in general and particularly on the SRI and Andorra execution models. We will extend the machine to support several lightweight processes per processor, allowing us to "hide" access delays with fast process switches.

11 RELATED WORK

The hardware organisation of the DDM was partly influenced by a proposal of Hermenegildo [5] to provide an address-escaping mechanism in clustered shared-memory architecture (essentially a hybrid between a shared memory machine and a message-passing machine). The DDM has many similarities to Wilson's proposal [10] for a hierarchical shared-memory architecture, and certain similarities to the Wisconsin Multicube [3]. However, all of these machines, unlike the DDM, depend on physically shared memory providing a "home" location for data. The Wisconsin Multicube can also be contrasted with the DDM in that certain requests need to be broadcast througout the entire machine.

12 ACKNOWLEDGEMENTS

DDM is part of the Esprit project PEPMA. We thank the many colleagues involved in or associated with the project. Especially David Warren of University of Bristol, who initiated many of the ideas, and who actively have been part of this work.

13 LIST OF STATES

I Invalid. The subsystem does not contain the item.

E Exclusive. This subsystem and no other contains the item.

S Shared. This subsystem and possibly other subsystems contain the item.

R Reading. This subsystem is waiting for a data value.

A Answering. This subsystem has promised to reply to a read request.

W Waiting. This subsystem is waiting to become exclusive.

L Leaving. This subsystem is about to get rid of this item.

14 DDM BUS

The DDM bus provides the following functionality:

Init : initiation of a transaction carrying the transaction code and the address of an item.

Data : carrying the data part of an item.

Ctr : carrying the counter value of a transaction.

Answer : the listening subsystems can answer yes. During this phase it will be determined if none or at least one answered yes.

Select : one of the subsystems answering yes is selected.

15 LIST OF TRANSACTION

Bus functionality of each transaction is listed in the parentethes:

r, read (init, answer, select). There is a request somewhere in the system to read this item

e, erase (init). Erase all your copies of this item.

d, data (init, data). A (possibly) shared copy of the item.

i, inject (init, data, ctr, answer, select). The one and only copy of an item is looking for a subsystem to move into.

o, out (init, data, answer). An item on its way out of the subsystem. It will stop when another copy of the item is found.

x, exclusive (init). Now there is only one copy of the item in the system.

l, leave (init, answer, select). This item should leave the subsystem.

u, up (init, data). Item about to leave the subsystem. It can only move up.

pr, physical-read (init). A read request on its way to a disk.

pw, physical-write (init,data). The last copy of an item on its way to a disk.

16 THE PROTOCOL TABLES

The tables define the DDM protocol. They describe how a controller responds to the different transactions according to the state in which the data item is when the transaction arrives. Each state has its own column and each signal its own row. Actions have the format: guard→NEWSTATE:transaction-to-send$_{Index}$, where index A means to the bus above and index B means to the bus below. An empty square means no action, the rest of the symbols are explained below.

$\emptyset$ This situation is impossible.
$\perp$ The processor may continue with its operation.
$a := yes$ The client is answering yes.
$a = 0 \rightarrow$ No client answered yes.
$a \geq 1 \rightarrow$ At least one client answered yes.
selected→ The client is answering yes. No other client finished the transaction; the client was selected during the selection phase.
X:y_{+1} The counter is incremented before the transaction y is sent.
$ctr > limit \rightarrow$ The counter has passed the limit.

MEMORY PROTOCOL FOR TRANSACTIONS BELOW					
Trans-action	**States** I	E	S	R	W
read	R:r_A[1]	$\perp$	$\perp$	$\emptyset$	$\emptyset$
write	read;write	$\perp$	W:e_A	$\emptyset$	$\emptyset$
replace[2]	$\emptyset$	I:i_A	I:o_A	$\emptyset$	$\emptyset$

[1] Preceded by a replace if no empty space in corresponding set.
[2] The replace first chooses an item to be replaced.

MEMORY PROTOCOL FOR TRANSACTIONS ABOVE					
Trans-action	**States** I	E	S	R	W
r		S:d_A	*selected*→S:d_A	R:-	W:-
e		$\emptyset$	I:-	R:r_A	I:write
d		$\emptyset$		S:$\perp$	
x		$\emptyset$	$\emptyset$	R:r_A	E:$\perp$
o		$\emptyset$	$a := yes$	$a := yes$ S[1]:$\perp$	$a := yes$
i	*selected*→[2] E:-	$\emptyset$	$\emptyset$	$a := yes$ S[1]:$\perp$	$\emptyset$
l		I:u_A	*selected*→I:u_A I:-		*selected*→I:u_A;write
u				R:r_A	I:write

[1] Might actually be exclusive.
[2] If empty space in the corresponding set.

DIRECTORY PROTOCOL FOR TRANSACTIONS BELOW							
Trans-action	States I	E	S	R	W	A	L
r	R:r_A[1]						
e		E:x_B	W:e_A	$\emptyset$	$\emptyset$	W:e_A	
d				$\emptyset$	$\emptyset$	S:d_A	
o		$a = 0 \rightarrow$E:i_B	$a = 0 \rightarrow$I:o_A	$\emptyset$	$\emptyset$	$a = 0 \rightarrow$I:o_A $a \geq 1 \rightarrow$S:d_A	
i[2]	$\emptyset$	$a = 0 \rightarrow$I:$i_{+1\,A}$	$\emptyset$	$\emptyset$	$\emptyset$	$a = 0 \rightarrow$I:o_A $a \geq 1 \rightarrow$A:r_B	$a = 0 \rightarrow$I:u_A $a \geq 1 \rightarrow$L:l_B
u		I:i_A	I:o_A	$\emptyset$	$\emptyset$	I:o_A	I:u_A

[1] If corresponding set is full, an item x is chosen to be replaced; then the following transactions are sent: $l(x)_B$; e_B.

[2] The transaction might be sent by the directory itself.

DIRECTORY PROTOCOL FOR TRANSACTIONS ABOVE							
Trans-action	States I	E	S	R	W	A	L
r		A:r_B	*selected*$\rightarrow$A:r_B			A:-	$a :=$ *yes*
e		$\emptyset$	I:e_B	R:r_A	I:e_B	I:e_B	I:e_B
d		$\emptyset$		S:d_B		S:-	
x		$\emptyset$	$\emptyset$	R:r_A	E:x_B	$\emptyset$	$\emptyset$
o		$\emptyset$	$a :=$ *yes*	$a :=$ *yes* S:d_B	$a :=$ *yes*	$a :=$ *yes* S:-	$a :=$ *yes*
i	*ctr* > *limit*$\rightarrow$I:- *selected*$\rightarrow$E:i_B	$\emptyset$	$\emptyset$	$a :=$ *yes* S:d_B	$\emptyset$	$\emptyset$	$\emptyset$
l		L:l_B	*selected*$\rightarrow$L:l_B I:e_B		*selected*$\rightarrow$L:l_B	*selected*$\rightarrow$L:l_B I:e_B	L:-
u		$\emptyset$	$\emptyset$	R:r_A	I:e_B		$\emptyset$

TOP DIRECTORY BELOW		
Trans-action	$a = 0 \rightarrow$	$a \geq 1 \rightarrow$
r	pr$_B$	
e	x$_B$	x$_B$
d		
x		
o	i$_B$	
i	pw$_B$	
l	$\emptyset$	$\emptyset$
u	$\emptyset$	$\emptyset$

INPUT BELOW FILTER		
Trans-action	$a = 0 \rightarrow$	$a \geq 1 \rightarrow$
r	r	
e	e	e
d	d	d
x		
o	o	o
i	i	i
l	l	l
u	u	u